# Министерство Образования Республики Беларусь

Учреждение образования

«Гомельский государственный технический университет

имени П.О.Сухого»

Кафедра: «Информатика»

по курсу: «Алгоритмы и структуры данных»

Лабораторная работа № 2

«Создание и обработка структур типа «дерево»»

Выполнил: студент группы ИП-32

Суховенко Э. С.

Допуск к защите: Проверил: преподаватель

Дата защиты: Процкая Мария Александровна

Гомель 2022 г.

**Цель:** Научиться создавать и обрабатывать структуры типа «дерево».

**Задание 1.**

Разработать проект для обработки дерева поиска, каждый элемент которого содержит структуру данных из варианта курсовой работы. Ключ поиска указан в варианте. В качестве структуры использовать 1) структуру с указателями на правого и левого потомка 2) массив с двумя дополнительными полями. В программе должны быть реализованы следующие возможности:

– создание дерева;

– добавление новой записи;

– поиск информации по заданному ключу;

– удаление информации с заданным ключом;

– вывод информации;

– решение индивидуального задания;

– освобождение памяти при выходе из программы.

**Индивидуальное задание:**

12. Найти среднее значение всех ключей дерева.

**Алгоритм решения задачи:**

Бинарное дерево:

Вставка:

1. используем рекурсивную функцию, в зависимости от вставляемого значения выбираем куда идти (налево если текущий элемент > вставляемого, направо если текущий элемент <= вставляемого)

2. продолжаем алгоритм, до тех пор пока не дойдём до состояния, когда место, в которое мы хотим вставить ничем не занято

3. создаём новую вершину и записываем ссылку на неё в предка, которого мы нашли на предыдущем шаге

Удаление:

1. выполняем сначала поиск вершины и запоминаем предка на каждом шаге

2. вершина является листом: помечаем ссылку на удаляемую вершину как null, освобождаем память

Поиск:

1. если текущее значение меньше цели поиска то идём направо, иначе налево

2. если пришли в вершину, где следующий шаг равен null, то возвращаем false, что значит что элемента не существует, иначе возвращаем true.

**Листинг программы:**

**ArrayBinaryTree.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

namespace lab2

{

class TreeBasedOnArray

{

public NodeTreeBasedOnArray Root { get; set; }

private NodeTreeBasedOnArray[] array = null;

public int \_Root { get; set; } = -1;

public NodeTreeBasedOnArray Make()

{

NodeTreeBasedOnArray prev = null, t;

bool find;

int key;

if (Root == null)

{

Console.Write("Введите значение ключа корня дерева: ");

key = int.Parse(Console.ReadLine());

Root = List(key);

array = new NodeTreeBasedOnArray[1];

array[array.Length - 1] = Root;

\_Root = 0;

}

Console.WriteLine("Добавление узлов дерева...");

while (true)

{

Console.Write("Введите ключ нового узла дерева: ");

key = int.Parse(Console.ReadLine());

if (key < 0) break;

t = Root;

find = false;

while (t != null && !find)

{

prev = t;

if (key == t.Key)

find = true;

else if (key < t.Key)

{

if (t.Left == int.MinValue)

t = null;

else

t = array[t.Left];

}

else

{

if (t.Right == int.MinValue)

t = null;

else

t = array[t.Right];

}

}

if (!find)

{

t = List(key);

Array.Resize(ref array, array.Length + 1);

array[array.Length - 1] = t;

if (key < prev.Key)

prev.Left = array.Length - 1;

else

prev.Right = array.Length - 1;

}

}

return Root;

}

public NodeTreeBasedOnArray List(int key)

{

return new NodeTreeBasedOnArray(key);

}

public void Watch(NodeTreeBasedOnArray t, int level = 0)

{

if (t != null)

{

if (t.Right == int.MinValue)

Watch(null, level + 1);

else

Watch(array[t.Right], level + 1);

for (int i = 0; i < level; i++)

Console.Write(" ");

Console.WriteLine("{0}", t.Key);

if (t.Left == int.MinValue)

Watch(null, level + 1);

else

Watch(array[t.Left], level + 1);

}

}

public NodeTreeBasedOnArray Search(int key)

{

NodeTreeBasedOnArray searchTreeNode, prevSearchTreeNode;

searchTreeNode = Root;

prevSearchTreeNode = null;

while (searchTreeNode != null && searchTreeNode.Key != key)

{

prevSearchTreeNode = searchTreeNode;

if (searchTreeNode.Key > key)

{

if (searchTreeNode.Left == int.MinValue)

searchTreeNode = null;

else

searchTreeNode = array[searchTreeNode.Left];

}

else

{

if (searchTreeNode.Right == int.MinValue)

searchTreeNode = null;

else

searchTreeNode = array[searchTreeNode.Right];

}

}

if (searchTreeNode == null)

{

Console.WriteLine("Узел дерева с таким ключом не найден!");

return Root;

}

else

{

Console.WriteLine("Узел дерева с заданным ключом найден");

return searchTreeNode;

}

}

public NodeTreeBasedOnArray Delete(int key)

{

NodeTreeBasedOnArray deletedNodeTree = Root, previousNodeTree = Root;

while (deletedNodeTree.Key != key && deletedNodeTree != null)

{

previousNodeTree = deletedNodeTree;

if (deletedNodeTree.Key > key)

deletedNodeTree = deletedNodeTree.Left == int.MinValue ? null : array[deletedNodeTree.Left];

else

deletedNodeTree = deletedNodeTree.Right == int.MinValue ? null : array[deletedNodeTree.Right];

}

if (deletedNodeTree == null)

{

Console.WriteLine("Узел с таким ключом не найден, удаление невозможно!");

return Root;

}

if (deletedNodeTree.Right == int.MinValue && deletedNodeTree.Left == int.MinValue)

{

if (deletedNodeTree == Root)

Root = null;

else if (array[previousNodeTree.Right] == deletedNodeTree)

{

previousNodeTree.Right = int.MinValue;

}

else

{

previousNodeTree.Left = int.MinValue;

}

}

else if (deletedNodeTree.Right == int.MinValue)

{

if (deletedNodeTree == Root)

Root = array[deletedNodeTree.Left];

else if (previousNodeTree.Right != int.MinValue && array[previousNodeTree.Right] == deletedNodeTree)

{

previousNodeTree.Right = deletedNodeTree.Left;

}

else

{

previousNodeTree.Left = deletedNodeTree.Left;

}

}

else if (deletedNodeTree.Left == int.MinValue)

{

if (deletedNodeTree == Root)

Root = array[deletedNodeTree.Right];

else if (previousNodeTree.Right != int.MinValue && array[previousNodeTree.Right] == deletedNodeTree)

{

previousNodeTree.Right = deletedNodeTree.Right;

}

else

{

previousNodeTree.Left = deletedNodeTree.Right;

}

}

else

{

if (Root == deletedNodeTree)

Root = array[deletedNodeTree.Right];

else if (previousNodeTree.Right != int.MinValue && array[previousNodeTree.Right] == deletedNodeTree)

{

previousNodeTree.Right = deletedNodeTree.Right;

}

else

{

previousNodeTree.Left = deletedNodeTree.Right;

}

NodeTreeBasedOnArray temp = array[deletedNodeTree.Right], insertedNodeTree = array[deletedNodeTree.Left];

while (true)

{

if (insertedNodeTree.Key > temp.Key)

{

if (temp.Right != int.MinValue)

temp = array[temp.Right];

else

{

temp.Right = Array.FindIndex(array, x => x == insertedNodeTree);

break;

}

}

else

{

if (temp.Left != int.MinValue)

temp = array[temp.Left];

else

{

temp.Left = Array.FindIndex(array, x => x == insertedNodeTree);

break;

}

}

}

}

int deleteIndex = Array.FindIndex(array, x => x == deletedNodeTree);

foreach (NodeTreeBasedOnArray node in array.Where(x => x.Left > deleteIndex))

node.Left -= 1;

foreach (NodeTreeBasedOnArray node in array.Where(x => x.Right > deleteIndex))

node.Right -= 1;

for (int i = deleteIndex; i < array.Length - 1; i++)

{

array[i] = array[i + 1];

}

Array.Resize(ref array, array.Length - 1);

Console.WriteLine("Узел с заданным ключом удален");

return Root;

}

}

}

**BinaryTree.cs:**

using System;

namespace lab2

{

class Tree

{

private NodeTree Root { get; set; }

public static int k = 0;

public NodeTree Make()

{

NodeTree prev = null, t;

bool find;

int key;

if (Root == null)

{

Console.Write("Введите значение ключа корня дерева: ");

key = int.Parse(Console.ReadLine());

Root = List(key);

}

Console.WriteLine("Добавление узлов дерева...");

while (true)

{

Console.Write("Введите ключ нового узла дерева (-1 если достаточно): ");

key = int.Parse(Console.ReadLine());

if (key < 0) break;

t = Root;

find = false;

while (t != null && !find)

{

prev = t;

if (key == t.Key)

find = true;

else if (key < t.Key)

t = t.Left;

else

t = t.Right;

}

if (!find)

{

t = List(key);

if (key < prev.Key)

prev.Left = t;

else

prev.Right = t;

}

}

return Root;

}

public NodeTree List(int key)

{

return new NodeTree(key);

}

public void Watch(NodeTree t, int level = 0)

{

if(t != null)

{

Watch(t.Right, level + 1);

for (int i = 0; i < level; i++) Console.Write(" ");

Console.WriteLine("{0}", t.Key);

Watch(t.Left, level + 1);

}

}

public NodeTree Search(int key)

{

NodeTree searchTreeNode, prevSearchTreeNode;

searchTreeNode = Root;

prevSearchTreeNode = null;

while(searchTreeNode != null && searchTreeNode.Key != key)

{

prevSearchTreeNode = searchTreeNode;

if (searchTreeNode.Key > key)

searchTreeNode = searchTreeNode.Left;

else

searchTreeNode = searchTreeNode.Right;

}

if(searchTreeNode == null)

{

Console.WriteLine("Узел дерева с таким ключом не найден!");

return Root;

}

else

{

Console.WriteLine("Узел дерева с заданным ключом найден");

return searchTreeNode;

}

}

public NodeTree Delete(int key)

{

NodeTree deletedNodeTree = Root, previousNodeTree = Root;

while(deletedNodeTree.Key != key && deletedNodeTree != null)

{

previousNodeTree = deletedNodeTree;

if (deletedNodeTree.Key > key)

deletedNodeTree = deletedNodeTree.Left;

else

deletedNodeTree = deletedNodeTree.Right;

}

if(deletedNodeTree == null)

{

Console.WriteLine("Узел с таким ключом не найден, удаление невозможно!");

return Root;

}

if(deletedNodeTree.Right == null && deletedNodeTree.Left == null)

{

if (deletedNodeTree == Root)

Root = null;

else if (previousNodeTree.Right == deletedNodeTree)

{

previousNodeTree.Right = null;

}

else

{

previousNodeTree.Left = null;

}

}

else if (deletedNodeTree.Right == null)

{

if (deletedNodeTree == Root)

Root = deletedNodeTree.Left;

else if (previousNodeTree.Right == deletedNodeTree)

{

previousNodeTree.Right = deletedNodeTree.Left;

}

else

{

previousNodeTree.Left = deletedNodeTree.Left;

}

}

else if (deletedNodeTree.Left == null)

{

if (deletedNodeTree == Root)

Root = deletedNodeTree.Right;

else if (previousNodeTree.Right == deletedNodeTree)

{

previousNodeTree.Right = deletedNodeTree.Right;

}

else

{

previousNodeTree.Left = deletedNodeTree.Right;

}

}

else

{

if (Root == deletedNodeTree)

Root = deletedNodeTree.Right;

else if (previousNodeTree.Right == deletedNodeTree)

{

previousNodeTree.Right = deletedNodeTree.Right;

}

else

{

previousNodeTree.Left = deletedNodeTree.Right;

}

NodeTree temp = deletedNodeTree.Right, insertedNodeTree = deletedNodeTree.Left;

while (true)

{

if (insertedNodeTree.Key > temp.Key)

{

if (temp.Right != null)

temp = temp.Right;

else

{

temp.Right = insertedNodeTree;

break;

}

}

else

{

if (temp.Left != null)

temp = temp.Left;

else

{

temp.Left = insertedNodeTree;

break;

}

}

}

}

Console.WriteLine("Узел с заданным ключом удален");

return Root;

}

public bool isLeaf(NodeTree node)

{

if (node.Left == null && node.Right == null)

{

return true;

}

return false;

}

public int SumValues(NodeTree node)

{

if (node == null) return 0;

return SumValues(node.Left) + SumValues(node.Right) + node.Key;

}

public int GetSize(NodeTree node)

{

if (node == null) return 0;

return GetSize(node.Left) + GetSize(node.Right) + 1;

}

public double CalcAverageValue(NodeTree root) => (double) SumValues(root) / GetSize(root);

public int height(NodeTree node)

{

if (node == null)

{

return 0;

}

else

{

int lheight = height(node.Left);

int rheight = height(node.Right);

if (lheight > rheight)

{

return (lheight + 1);

}

else

{

return (rheight + 1);

}

}

}

}

}

*Асимптотическая оценка сложности алгоритма:*

Вставка ­­― O(h), где h – высота дерева, потому что вставляемая вершина занимает одно положение в дереве, а дойти до этого положения потребует максимально h шагов

Удаление ­­― O(h), где h – высота дерева, аналогично вставке

Поиск ­­― O(h), где h – высота дерева, аналогично вставке

*Наилучшую, наихудшую и среднюю оценки:*

Наилучшая оценка всех операций: O(1), если мы работаем с корнем

Наихудшая оценка: O(n), где n – кол-во вершин в дереве и дерево представляет собой список, имеется в виду, что все вершины распологаются только в левых поддеревьях друг друга либо только в правых и тогда мы теряем основное преимущество дерева

Средняя оценка: O(h) для несбалансированного дерева или O(log2(n)), если дерево сбалансированное.

*Псевдокод операции на упрощённом языке:*

Вставка:

insert(vertex, value, parent = vertex):

if (parent == null || vertex == null) return new TreeNode(value)

if (vertex.value > value) vertex.left = insert(vertex.left, value, vertex)

else vertex.right = insert(vertex.right, value, vertex)

return vertex

// example

root = null

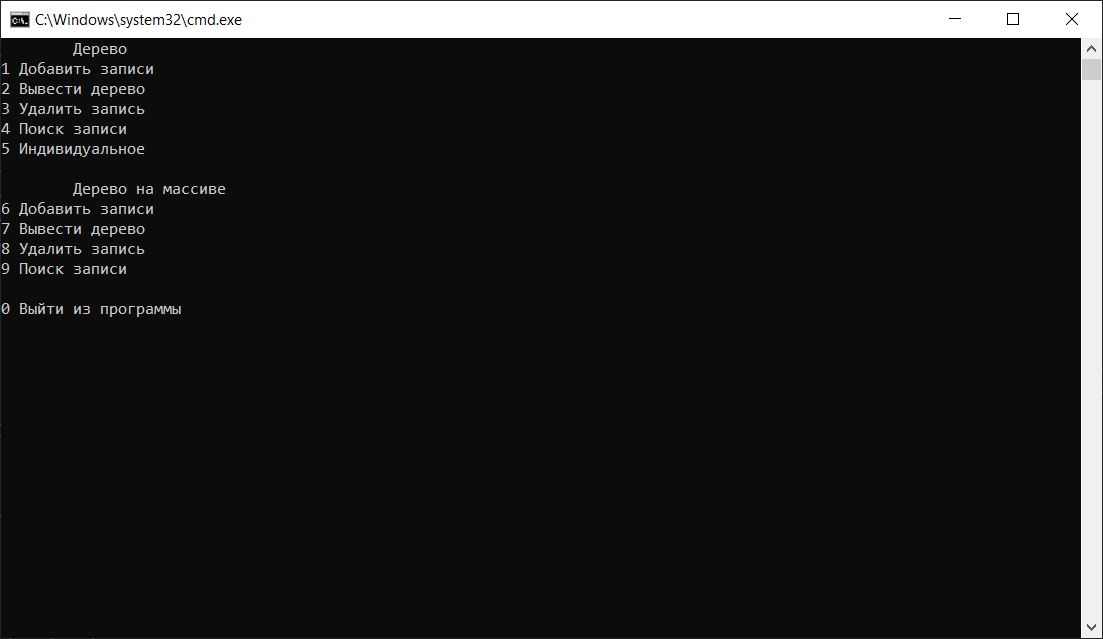
root = insert(root, 5)

root = insert(root, 13)

*Выкладки расчета асимптотической оценки:*

Приведены в пункте “Асимптотическая оценка”.

*Выкладки или результаты тестирования на различных по объему набору данных:*



**Tree:**

**2**

**4**

**3**

**2**

**1**

**Average value of tree: 2.5**

**Tree:**

**5**

**4**

**3**

**1**

**Average value of tree: 3.25**

**Вывод:** изучил структуры типа «дерево».